Université Abdelmalek Essaâdi
École Normale Supérieure
Tétouan
Département d'Informatique et de Gestion

Chapitre 11

La couche métier : les entités

LP-DW

2017-2018

Notions d'ORM

- **L'objectif d'un ORM** (Object-Relation Mapper) « lien objet-relation » :
  - Se charger de l'enregistrement des données en faisant oublier qu'on a une base de données.
  - On va plus écrire de requêtes, ni créer de tables via phpMyAdmin.
  - Dans le code PHP, on va faire appel à Doctrine2, l'ORM par défaut de **Symfony**, pour faire tout cela.

Notions d'ORM : fini les requêtes, utilisons des objets

- **Exemple** :
  - Une variable $utilisateur contenant un objet User , qui représente l'un des utilisateurs qui vient de s'inscrire sur un site.
  - Pour sauvegarder cet objet, on crée une fonction qui effectue une requête SQL du type INSERT INTO dans la bonne table, etc.
  - On doit gérer tout ce qui touche à l'enregistrement en base de données.
- **En utilisant un ORM** :
  - On n'aura plus qu'à utiliser quelques fonctions de cet **ORM :** $orm->save($utilisateur).
  - On peut enregistrer l'utilisateur en une seule ligne.
- Pour bien utiliser un ORM : Oubliez les requêtes SQL, pensez objet

les données sont des objets

- Accéder à des attributs via $utilisateur['email']
- **Utiliser des objets** : Faire $utilisateur->getEmail() au lieu de $utilisateur['email']

- **Une révolution** : coupler cette représentation objet avec l'ORM.
- **Exemple** : $utilisateur->getCommentaires()
  - la méthode $utilisateur->getCommentaires() déclencherait la bonne requête, récupérerait tous les commentaires postés par l'utilisateur, et retournerait **un tableau d'objets** de type Commentaire qu'on pourrait afficher sur la page de profil de l' utilisateur.

Vocabulaire

- Un objet dont vous confiez l'enregistrement à l'ORM s'appelle une **entité** (**entity** en anglais).
- On dit également persister une **entité,** plutôt qu'enregistrer une **entité.**

Créer une première entité avec Doctrine2

Une entité, c'est juste un objet

- Une entité, ce que l'ORM va manipuler et enregistrer dans la base de données, ce n'est vraiment rien d'autre qu'un simple objet.
- l'objet Advert de la plateforme d'annonces :

Une en

- Une entité, c
- l'objet Adve

de

données, ce

```php
<?php
// src/DW/PlatformBundle/Entity/Advert.php
namespace DW\PlatformBundle\Entity;
class Advert {
        protected $id;
        protected $content;
        // Et bien sûr les getters/setters :
        public function setId($id)  {
                $this->id = $id;
        }
        public function getId()  {
                return $this->id;
        }
        public function setContent($content)  {
                $this->content = $content;
        }
        public function getContent()  {
                return $this->content;
        }
}
```

Une entité, c'est juste un objet

- Utiliser l'objet

```php
<?php
// src/DW/PlatformBundle/Controller/AdvertController.php
namespace DW\PlatformBundle\Controller;
use DW\PlatformBundle\Entity\Advert;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
class AdvertController extends Controller
{
    public function viewAction()
    {
        $advert = new Advert;
        $advert->setContent("Recherche développeur Symfony3.");
        return $this->render('DWPlatformBundle:Advert:view.html.twig', array(
        'advert' => $advert
        ));
    }
}
```

Une entité, c'est juste un objet

- Comment l'ORM va-t-il faire pour enregistrer cet objet dans la base de données s'il ne connaît rien des propriétés id et content ?
- Comment peut-il deviner que la propriété id doit être stockée dans une colonne de type INT dans la table ?

Une entité, c'est juste un objet

- Comment l'ORM va-t-il faire pour enregistrer cet objet dans la base de données s'il ne connaît rien des propriétés id et content ?
- Comment peut-il deviner que la propriété id doit être stockée dans une colonne de type INT dans la table ?

il ne devine rien, on va le lui dire !

Une entité, c'est juste un objet… mais avec des commentaires

- On va ajouter des **commentaires** dans le code et Symfony va se servir directement de ces **commentaires** pour ajouter des fonctionnalités à l'application.
- Ce type de commentaires se nomme l'annotation.
- Les annotations doivent respecter une syntaxe particulière :

Une entité, c'est juste un objet… mais avec des commentaires

```php
<?php
// src/DW/PlatformBundle/Entity/Advert.php
namespace DW\PlatformBundle\Entity;
// On définit le namespace des annotations utilisées par Doctrine2
// En effet, il existe d'autres annotations,
// qui utiliseront un autre namespace
use Doctrine\ORM\Mapping as ORM;
/**
* @ORM\Entity
*/
class Advert
{
  /**
  * @ORM\Column(name="id", type="integer")
  * @ORM\Id
  * @ORM\GeneratedValue(strategy="AUTO")
  */
  protected $id;
```

- e code et Symfony va se servir
- ajouter des fonctionnalités à
- notation.
- ntaxe particulière :

Une entité, c'est juste un objet… mais avec des commentaires

```php
<?php
// src/DW/PlatformBundle/Entity/Advert.php
namespace DW\PlatformBundle\Entity;
// On définit le namespace des annotations utilisées par Doctrine
// En effet, il existe d'autres annotations,
// qui utiliseront un autre namespace
use Doctrine\ORM\Mapping as ORM;
/**
* @ORM\Entity
*/
class Advert
{
  /**
  * @ORM\Column(name="id", type="integer")
  * @ORM\Id
  * @ORM\GeneratedValue(strategy="AUTO")
  */
  protected $id;
/**
* @ORM\Column(name="date", type="date")
*/
protected $date;
/**
* @ORM\Column(name="title", type="string", length=255)
*/
protected $title;
/**
* @ORM\Column(name="author", type="string", length=255)
*/
protected $author;
/**
* @ORM\Column(name="content", type="text")
*/
protected $content;
// Les getters
// Les setters
}
```

Une entité, c'est juste un objet… mais avec des commentaires

- Ne recopiez toujours pas toutes ces annotations à la main,
  - on utilise le générateur en console
- **Attention :**
  - Pour les prochaines annotations que vous serez amenés à écrire à la main : elles doivent être dans des commentaires de type «/**», avec précisément deux étoiles.
  - Si vous essayez de les mettre dans un commentaire de type «/*» ou encore «//», elles seront simplement ignorées.

Une entité, c'est juste un objet… mais avec des commentaires

- Grâce à ces annotations, Doctrine2 dispose de toutes les informations nécessaires pour :
  - utiliser l'objet,
  - créer la table correspondante,
  - l'enregistrer,
  - définir un identifiant (id) en auto-incrément,
  - nommer les colonnes,
  - etc.
- Ces informations se nomment les metadata de l'entité.

Une entité, c'est juste un objet… mais avec des commentaires

- rajouter les metadata à l'objet Advert : s'appelle mapper l'objet Advert.
  - C'est-à-dire faire le **lien** entre l'objet et la représentation physique qu'utilise Doctrine2 (une table SQL).
- Il existe d'autres moyens de définir les metadata d'une entité : en YAML, en XML et en PHP.

Générer une entité

- Il faut créer une base de données avant de lancer les commandes qui vont suivre :
  - Pour pouvoir lancer les commandes provenant de DoctrineBundle, la librairie vérifie la connexion à la base de données configurée.
  - Si la base de données n'existe pas, vous ne pourrez malheureusement pas lancer les prochaines commandes.
- Il faut donc rendre dans le fichier app/config/parameters.yml et adapter les paramètres propre à la base de données (utilisateur, mot de passe, nom de la base de données…), puis lancez la commande de création de la base de données :

```
C:\wamp\www\Symfony>php bin/console doctrine:database:create
```

Générer une entité

- la génération de l'entité :

```
C:\wamp\www\Symfony>php bin/console doctrine:generate:entity
```

Générer une entité

- **Première étape, le nom :**

```
Welcome to the Doctrine2 entity generator


This command helps you generate Doctrine2 entities.

First, you need to give the entity name you want to generate.
You must use the shortcut notation like AcmeBlogBundle:Post.

The Entity shortcut name:_
```

- il faut entrer le nom de l'entité sous le format NomBundle:NomEntité.
- Dans notre cas, on entre donc DWPlatformBundle:Advert.

Générer une entité

- **Configuration :**

```
The Entity shortcut name: DWPlatformBundle:Advert

Determine the format to use for the mapping information.

Configuration format (yml, xml, php, or annotation)
[annotation]:_
```

- On va utiliser les annotations, qui sont d'ailleurs le format par défaut.
- Appuyez juste sur la touche Entrée

Générer une entité

- **Définition d'un champ :**

```
Configuration format (yml, xml, php, or annotation) [annotation]:

Instead of starting with a blank entity, you can add some fields now.
Note that the primary key will be added automatically (named id).

Available types: array, simple_array, json_array, object,
boolean, integer, smallint, bigint, string, text, datetime, datetimetz,
date, time, decimal, float, blob, guid.

New field name (press <return> to stop adding fields):_
```

- saisir le nom de des champs.
- Lisez bien ce qui est inscrit avant :
  - Doctrine2 va ajouter automatiquement l'id, de ce fait, pas besoin de le redéfinir ici. On entre date.

Générer une entité

- **Définition du type du champ :**

```
New field name (press <return> to stop adding fields):
date
Field type [string]:_
```

- On va dire à Doctrine à quel type correspond la propriété date : datetime

Générer une entité

- Le champ est-il facultatif ?

```
Field type [string]: datetime
Is nullable [false]:
```

- le champ date n'est pas facultatif, on répond donc false, la valeur par défaut, en tapant sur Entrée.

Générer une entité

- Le champ est-il unique ?

```
Is nullable [false]:
Unique [false]:
```

- Les champs sont peu souvent uniques, en tout cas ce n'est pas le cas du champ date, tapez donc Entrée.

Générer une entité

- Répétez les 4 derniers points pour les propriétés title , author et content .
- title et author sont de type string de 255 caractères.
- Content est de type text.

Générer une entité

- Répétez les 4 derniers points pour le
- title et author sont de type string
- Content est de type text.

```
New field name (press <return> to stop adding fields): date
Field type [string]: datetime
Is nullable [false]:
Unique [false]:

New field name (press <return> to stop adding fields): title
Field type [string]:
Field length [255]:
Is nullable [false]:
Unique [false]:

New field name (press <return> to stop adding fields): author
Field type [string]:
Field length [255]:
Is nullable [false]:
Unique [false]:

New field name (press <return> to stop adding fields): content
Field type [string]: text
Is nullable [false]:
Unique [false]:

New field name (press <return> to stop adding fields):_
```

Générer une entité

- Lorsque vous avez fini, appuyez sur la touche Entrée.

```
Entity generation


> Generating entity class D:\www\Symfony\src\DW\PlatformBundle\Entity\Advert.php: OK!
> Generating repository class D:\www\Symfony\src\DW\PlatformBundle\Repository\AdvertRepository.php: OK!
```

Générer une entité

- le résultat dans le fichier : Entity/Advert.php.
- Symfony a tout généré, même les **getters** et les **setters** !
  - une simple classe… avec plein d'annotations.

Affiner l'entité avec de la logique métier

En plus de gérer les données un modèle contient également la logique de l'application.

- **Exemple 1: Attributs calculés**
  - une entité Commande, qui représenterait un ensemble de produits à acheter sur un site d'e-commerce.
  - Cette entité aurait les attributs suivants :
    - ListeProduits qui contient un tableau des produits de la commande ;
    - AdresseLivraison qui contient l'adresse où expédier la commande ;
    - Date qui contient la date de la prise de la commande ;
    - Etc.

Affiner l'entité avec de la logique métier

- Ces trois attributs devront bien entendu être *mappés* (c'est-à-dire définis comme des colonnes pour l'ORM via des annotations) pour être enregistrés en base de données par Doctrine2.
- Il existe d'autres caractéristiques pour une commande, qui nécessitent un peu de calcul : le prix total, un éventuel coupon de réduction, etc.
- Ces caractéristiques n'ont pas à être persistées en base de données, car elles peuvent être déduites des informations que l'on a déjà.
- Par exemple, pour avoir le prix total, il suffit de faire une boucle sur ListeProduits et d'additionner le prix de chaque produit :

Affiner l'entité avec de la logique métier

- Ces trois attr… dire définis comme des co… registrés en base de donn…
- Il existe d'aut… itent un peu de calcul : le p…
- Ces caractéris… es, car elles peuvent être …
- Par exemple, … boucle sur ListeProduits e…

```php
<?php
// Exemple :
class Commande
{
    public function getPrixTotal()
    {
        $prix = 0;
        foreach($this->getListeProduits() as $produit) {
        $prix += $produit->getPrix();
        }
        return $prix;
    }
}
```

Affiner l'entité avec de la logique métier

- Créer des méthodes getQuelquechose() qui contiennent de la logique métier.
- L'avantage de mettre la logique dans l'entité même est qu'on est sûrs de réutiliser cette même logique partout dans l'application.
- Il est bien plus propre et pratique de faire $commande->getPrixTotal() pour calculer le prix total.

- Ces méthodes n'ont pas d'équivalent setQuelquechose()

Affiner l'entité avec de la logique métier

- **Exemple 2: Attributs par défaut**
  - Si on a besoin de définir une certaine valeur à des entités lors de leur création.
    - Or des entités sont de simples objets PHP, et la création d'un objet PHP fait appel… au constructeur.
  - Pour l'entité Advert, on pourrait définir le constructeur suivant :

```php
<?php
// src/DW/PlatformBundle/Entity/Advert.php
namespace DW\PlatformBundle\Entity;
use Doctrine\ORM\Mapping as ORM;
/**
* Advert
* @ORM\Table()
* @ORM\Entity(repositoryClass= "DW\PlatformBundle\Entity\AdvertRepository")
*/
class Advert
{
    // ...
    public function __construct()
    {    // Par défaut, la date de l'annonce est la date d'aujourd'hui
        $this->date = new \Datetime();
    }
    // ...
}
```

Entité

- Une entité est un objet PHP qui correspond à un besoin dans l'application.
- N'essayez donc pas de raisonner en termes de tables, base de données, etc.
- Vous travaillez maintenant avec des objets PHP, qui contiennent une part de logique métier, et qui peuvent se manipuler facilement.

Le mapping

Le mapping

- On a vu comment mapper les objets avec les annotations.
- Ces annotations permettent d'inscrire pas mal d'autres informations.
- Il faut juste en connaître la syntaxe, c'est l'objectif de cette section.
- Tout ce qu'on va voir se trouve bien entendu dans la documentation officielle sur le mapping :

http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/basic-mapping.html

L'annotation Entity

- L'annotation Entity s'applique sur une classe, il faut donc la placer avant la définition de la classe en PHP.
- Elle définit un objet comme étant une entité, et donc persisté par Doctrine.
- Cette annotation s'écrit comme suit : `@ORM\Entity`
- Deux paramètres sont possibles pour cette annotation :
  - repositoryClass
  - readOnly

repositoryClass

- Il permet de préciser le namespace complet du repository qui gère cette entité.
- On donnera le même nom aux repositories qu'aux entités, en les suffixant simplement de "Repository".
- Pour l'entité Advert :

```
@ORM\Entity(repositoryClass=« DW\PlatformBundle\Entity\AdvertRepository")
```

readOnly

- si vous ajoutez cette annotation pour une entité, on demande à Doctrine ne plus tracker les modifications qui peuvent survenir sur l'objet.
- il est toujours possible d'ajouter (méthode persist()) ou supprimer (méthode remove()) un objet.

```
@ORM\Entity(readOnly=true)
```

L'annotation Table

- L'annotation Table s'applique sur une classe.
- C'est une annotation facultative, une entité se définit juste par son annotation Entity.
  - l'annotation Table permet de personnaliser le nom de la table qui sera créée dans la base de données.
  - Par exemple, on pourrait préfixer notre table advert par « dw » :

```
@ORM\Table(name=« dw_advert")
```

  - Elle se positionne juste avant la définition de la classe.
- Par défaut, si on ne précise pas cette annotation, le nom de la table créée par Doctrine2 est le même que celui de l'entité.

L'annotation Column

- L'annotation Column s'applique sur un attribut de classe, elle se positionne donc juste avant la définition PHP de l'attribut concerné.
- Cette annotation permet de définir les **caractéristiques** de la colonne concernée :

```
@ORM\Column
```

- L'annotation Column comprend quelques paramètres, dont le plus important est le type de la colonne.

Les types de colonnes

- Les types de colonnes qu'on peut définir en annotation sont des types Doctrine, et uniquement Doctrine.
- Ne les confondez pas avec leurs homologues SQL ou PHP, ce sont des types à Doctrine seul.
- Ils font la transition des types SQL aux types PHP.

Les types de colonnes

- la liste exhaustive des types Doctrine2 disponibles.

Type Doctrine	Type SQL	Type PHP	Utilisation
string	VARCHAR	string	Toutes les chaînes de caractères jusqu'à 255 caractères.
integer	INT	integer	Tous les nombres jusqu'à 2 147 483 647.
smallint	SMALLINT	integer	Tous les nombres jusqu'à 32 767.
bigint	BIGINT	string	Tous les nombres jusqu'à 9 223 372 036 854 775 807. Attention, PHP reçoit une chaîne de caractères, car il ne supporte pas un si grand nombre (suivant que vous êtes en 32 ou en 64 bits).
boolean	BOOLEAN	boolean	Les valeurs booléennes true et false.
decimal	DECIMAL	double	Les nombres à virgule.
date ou datetime	DATETIME	objet DateTime	Toutes les dates et heures.
time	TIME	objet DateTime-	Toutes les heures.
text	CLOB	string	Les chaînes de caractères de plus de 255 caractères.
object	CLOB	Type de l'objet stocké	Stocke un objet PHP en utilisant serialize/unserialize.
array	CLOB	array	Stocke un tableau PHP en utilisant serialize/unserialize.
float	FLOAT	double	Tous les nombres à virgule. Attention, fonctionne uniquement sur les serveurs dont la locale utilise un point comme séparateur.

Les types de colonnes

- Les types Doctrine sont sensibles à la casse.
- Le type « String » n'existe pas, il s'agit du type « string ».
  - tout est en minuscule
- Le type de colonne se définit en tant que paramètre de l'annotation Column, comme suit :

```
@ORM\Column(type="string")
```

Les paramètres de l'annotation Column

- Il existe 7 paramètres, tous facultatifs, que l'on peut passer à l'annotation Column afin de personnaliser le comportement.
- La liste exhaustive dans le tableau suivant.

Paramètre	Valeur par défaut	Utilisation
type	string	Définit le type de colonne comme nous venons de le voir.
name	Nom de l'attribut	Définit le nom de la colonne dans la table. Par défaut, le nom de la colonne est le nom de l'attribut de l'objet, ce qui convient parfaitement. Mais vous pouvez changer le nom de la colonne, par exemple si vous préférez « isExpired » en attribut, mais « is_expired » dans la table.
length	255	Définit la longueur de la colonne. Applicable uniquement sur un type de colonne string.
unique	false	Définit la colonne comme unique. Par exemple sur une colonne e-mail pour vos membres.
nullable	false	Permet à la colonne de contenir des NULL.
precision	0	Définit la précision d'un nombre à virgule, c'est-à-dire le nombre de chiffres en tout. Applicable uniquement sur un type de colonne decimal.
scale	0	Définit le *scale* d'un nombre à virgule, c'est-à-dire le nombre de chiffres après la virgule. Applicable uniquement sur un type de colonne decimal.

Les types de colonnes

- Pour définir plusieurs options en même temps, il faut simplement les séparer avec une virgule.
- Par exemple, pour une colonne « e-mail » en string 255 et unique, il faudra faire :

```
@ORM\Column(type="string", length=255, unique=true)
```

Pour conclure

- la couche **Modèle** sous Symfony : l'utilisation des entités de l'ORM Doctrine2.
- l'adresse de la documentation Doctrine2 :

  http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/index.html
- Doctrine étant une bibliothèque totalement indépendante de Symfony, sa documentation fait référence à ce type d'annotation : /** @Entity **/
- Il faut impérativement l'adapter à votre projet Symfony, en préfixant toutes les annotations par « ORM\ » : /** @ORM\Entity **/
  - Car dans les entités, c'est le namespace ORM qu'on charge.
  - l'annotation @Entity n'existe pas, c'est @ORM qui existe (et tous ses enfants : @ORM\Entity, @ORM\Table, etc.).

En résumé

- Le rôle d'un ORM est de se charger de la persistance de vos données : vous manipulez des objets, et lui s'occupe de les enregistrer en base de données.
- L'ORM par défaut livré avec Symfony est Doctrine2.
- L'utilisation d'un ORM implique un changement de raisonnement : on utilise des objets, et on raisonne en POO. C'est au développeur de s'adapter à Doctrine2, et non l'inverse !
- Une entité est, du point de vue PHP, un simple objet. Du point de vue de Doctrine, c'est un objet complété avec des informations de mapping qui lui permettent d'enregistrer correctement l'objet en base de données.
- Une entité est, du point de vue de votre code, un objet PHP qui correspond à un besoin, et indépendant du reste de votre application.